HARS: a Heterogeneity-Aware Runtime System for Self-Adaptive Multithreaded Applications

Jaeyoung Yun

Department of Electrical and Computer Engineering

(Computer Science and Engineering)

Graduate School of UNIST

2016

HARS: a Heterogeneity-Aware Runtime System for Self-Adaptive Multithreaded Applications

Jaeyoung Yun

Department of Electrical and Computer Engineering

(Computer Science and Engineering)

Graduate School of UNIST

HARS: a Heterogeneity-Aware Runtime System for Self-Adaptive Multithreaded Applications

A thesis submitted to the Graduate School of UNIST

in partial fulfillment of the requirements for

the degree of Master of Computer Science and Engineering

Jaeyoung Yun

12. 17. 2015

Approved by

Advisor

Woongki Baek

HARS: a Heterogeneity-Aware Runtime System for Self-Adaptive Multithreaded Applications

Jaeyoung Yun

This certifies that the thesis of

Jaeyoung Yun is approved.

12. 17. 2015

signature

Advisor: Woongki Baek

signature

Jongeun Lee : Thesis Committee Member #1

signature

Young-ri Choi: Thesis Committee Member #2

Abstract

We are in the age of rapid changes. In particular, the computer is not just a device for carrying out complex computations. Various users can use these devices in various ways. Someone could just use a computer for document writing, while others can use one for video encoding or playing computer games. Each process has different computational demands. If we use a supercomputer for writing documents only, it is truly a wasteful mistake. It is like using a sledgehammer to crack a nut. Each process has own computational demands. If we can give the proper computation power to each process, this would be more sufficient than the before. In this manner, heterogeneous multi-processing (HMP) arose. HMP is a promising technique that can support both high and low demand tasks efficiently. This topic has been investigated in some prior works but an efficient system software to support HMP with self-adaptive computing has been little researched, especially on multithreaded applications. Therefore, we propose HARS, a heterogeneity-aware runtime system for self-adaptive multithreaded applications. HARS monitors application-level performance data and dynamically controls the system state to achieve the performance target with efficient power consumption. As an extended version of HARS, we also propose MP-HARS, which that supports multiple applications. Through our evaluation, we can see that HARS and MP-HARS achieve higher efficiency then the baseline version and HARS is comparable to the static optimal version.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Heterogeneous multi-processing (HMP) is an emerging technique that is used in many cutting-edge devices. HMP consists of various type of cores. For example, ARM big.LITTLE architecture [2] is the representative of this technique. This HMP architecture has two types of cores. The big core has high performance with high power consumption whereas, the LITTLE core has low performance with a power efficient core. The big core is proper to dealing with demanding work, while the LITTLE core is appropriate for low-demand work. This architecture was researched from prior works [8, 12, 15]. These works use system-level performance data (e.g., cache misses, number of cycles, cycles per instructions, CPU utilization) to achieve the system user requirements with efficiency in a HMP system. But these performance data cannot represent the specific performance of application. So if the manager monitors the system-level performance, it can lead the suboptimal result.

To adjust the system to an efficient state, self-adaptive computing is needed. Self-adaptive computing periodically monitors the application-level performance and controls the system to achieve efficiency [4]. This has also been investigated many times in prior work[4, 5, 6, 13], but they did not consider HMP or only considered a single-threaded application to achieve high efficiency [10]. Therefore, efficient system software supporting HMP with self-adaptive computing needs to be extensively investigated, especially in multithreaded-applications.

In this context, we propose HARS, a heterogeneity-aware runtime system for self-adaptive multi-threaded applications. HARS can set the target performance of the application. And, HARS periodically monitors application-level performance data and dynamically controls the system state to achieve the performance target with efficient power consumption.

HARS is designed to support a single multithreaded application. However, in the real world, the system runs multiple applications concurrently. Therefore, HARS needs to support multiple applications in an efficient manner. As an extended version of HARS, we also propose MP-HARS, which can support multiple applications. MP-HARS partitions the system resources to each application. The components that cannot be divided are dealt with independently. Our contributions from this work can be summarized in four points.

- We propose HARS, a heterogeneity-aware runtime system for self-adaptive multithreaded applications. HARS consists of three components, which are performance estimator, power estimator, and runtime manager. The runtime manager periodically monitors the application performance and leads the system state to achieve more efficiency by using two estimators.
- We design and implement a prototype of HARS, which is a user-level runtime system for commodity Linux-based systems. There is no requirement to modify the Linux kernel. HARS is implemented using four simple iterative loops to sweep the configurable system spaces. It also has two types of schedulers that leverage constructive cache sharing among consecutive threads or mitigate the performance imbalance of each thread.
- We propose MP-HARS, an extended version of HARS that can support multiple applications. MP-HARS adds two modules to HARS, one is the resource partitioning and the other is the interference-aware adaptation. These modules decide adaptation behavior for each application to approach a more efficient system state but do not affect the other applications to violate the target performance.
- We quantify the effectiveness of HARS using multithreaded PARSEC benchmarks [1] on a full HMP system. Our evaluation shows that HARS significantly outperforms the baseline version

and is comparable to the static optimal version. MP-HARS also shows significantly better results compared to the baseline version.

Chapter 2

Background

2.1 Heterogeneous multi-processing

HMP is an emerging technique that consists of multiple type of cores to achieve various performance requirements of applications with low power consumption. ARM's big.LITTLE processor [7] is a typical HMP architecture that is implemented on various cutting-edge mobile devices. This architecture consists of two type of cores. One is *big core* and the other is *little core*. Big core is relatively faster and more power hungry than little core. Little core is slower and uses less power. Big core is suitable for running demanding tasks, such as playing games, movies or navigation. This core type is usually complex and involves out-of-order architecture. The little core is suitable for low-demand tasks, such as listening to music, sending a message and phone calling. This core type is usually simple and uses in-order architecture. HMP architecture is managed by the Linux HMP scheduler. The latest version of this scheduler is the global task scheduler (GTS), which can migrate tasks to any type of cores based on the migration threshold.

2.2 Self-Adaptive Computing

Self-adaptive computing enables applications to set user-specified goals and dynamically controls the system to achieve user satisfaction with low power consumption [4]. Self-adaptive computing has three processes – (1) observation, (2) decision, and (3) action.

The *observation* component periodically checks if the current performance of the application satisfies the target performance. Performance information can be monitored by system-level data, such as CPU utilization and cycle per instruction. However, this information cannot show the real application performance data. Fortunately, the Application Heartbeats framework [4] can solve this problem. This framework enables the application to emit a *heartbeat* when it finishes its unit of work. From the heartbeat rate, we can know the application-level performance data.

The *decision* component searches the system space and chooses the best system state to adjust the system to satisfy the target with maximum efficiency. Efficiency means that it achieves the target performance with low power consumption. The *action* component sets the system state to be determined by the decision component.

Chapter 3

HARS

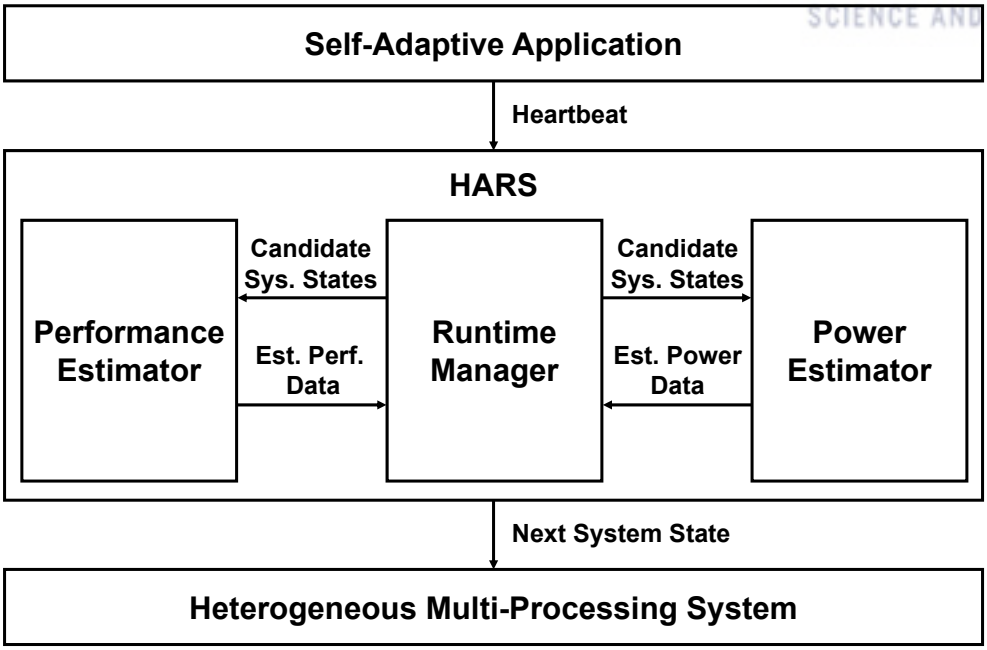


Figure 3.1: The overall architecture of HARS

3.1 HARS Design & Implementation

Figure 3.1 shows the overall architecture of HARS. HARS consists of three components, the performance estimator, the power estimator, and the runtime manager. When the self-adaptive application finishes its unit of work, it will emit a heartbeat and the runtime manager can know the application-level performance from the heartbeat rate. The runtime manager checks if the current performance satisfies the target performance. If it does not satisfy the target, it will generate the neighbor candidate states and estimate each state's performance and power consumption. The runtime manager then compares each state by efficiency metric (performance/power) and chooses the system state that has the best efficiency. HARS assumes that HMP system consists of two core types, but we believe that HARS can support more core types.

3.1.1 Performance Estimator

The *performance estimator* estimates the performance of the target self-adaptive multithreaded application. It assumes that the performance is proportional to the number of cores and the frequency level. It also determines the best thread assignment to maximize the performance of the application with the assigned cores. It tries to assign the threads to a faster core cluster first and the rest of threads are assigned to a slower core cluster. We assume that T is the total number of threads of the target application. Total work is equally distributed on each thread. T_B is the total number of threads in the big core cluster and T_L is the total number of threads in the little core cluster. So, we can say that $T = T_B + T_L$.

Let S_{B,f_0} and S_{L,f_0} be the per-core performance of the big and little cores, respectively. f_0 means the baseline frequency.[1] For example, if the big core is running at f_B frequency, then $S_B = \frac{f_B}{f_0} \cdot S_{B,f_0}$. If the little core is running at f_L frequency, then $S_L = \frac{f_L}{f_0} \cdot S_{L,f_0}$. The performance ratio between the big and little cores can be calculated as $r = \frac{S_B}{S_L}$.

[1] This work assumes that the underlying HMP system supports per-cluster DVFS not per-core DVFS.

T	T_B	T_L	$C_{B,U}$	$C_{L,U}$
$0 < T \leq C_B$	T	0	T	0
$C_B < T \leq r \cdot C_B$	T	0	C_B	0
$r \cdot C_B < T \leq r \cdot C_B + C_L$	$\lfloor r \cdot C_B \rfloor$	$T - T_B$	C_B	$T - T_B$
$r \cdot C_B + C_L < T$	$\left\lceil \frac{r \cdot C_B}{r \cdot C_B + C_L} \cdot T \right\rceil$	$T - T_B$	C_B	C_L

Table 3.1: Thread assignment to the big and little clusters ($r \geq 1$)

We define C_B and C_L as the number of big core and the number of little core that are allocated to the target self-adaptive application. Let t_B and t_L be the time spent in each threads T_B and T_L to complete a unit of work that uses C_B and C_L. Below shows how t_B and t_L are calculated.

$$t_B = \begin{cases} \frac{W_T}{S_B} & \text{if } T_B \leq C_B \\ \frac{T_B \cdot W_T}{C_B \cdot S_B} & \text{otherwise} \end{cases} \qquad t_L = \begin{cases} \frac{W_T}{S_L} & \text{if } T_L \leq C_L \\ \frac{T_L \cdot W_T}{C_L \cdot S_L} & \text{otherwise} \end{cases}$$

And, t_f is the total execution time to complete all of the work. t_f can be calculated as $t_f = \max(t_B, t_L)$.

Table 3.1 is the thread assignment table to minimize the t_f when $r \geq 1$. Note that $C_{B,U}$ and $C_{L,U}$ are the number of cores that are really used by the application and these can be smaller than the number of allocated cores (i.e., C_B and C_L). The results with $r < 1$ can be similarly derived.

3.1.2 Power Estimator

The *power estimator* estimates the power consumption of the target self-adaptive multithreaded application. It estimates the power consumption using the linear regression models that are constructed from the profiled data. This data is collected by the microbenchmark, which stresses the cores and memory with running tasks. This microbenchmark can configure the number of cores, frequency level, and CPU utilization. We assume that the power consumption of each cluster is proportional to the number of used cores in each cluster and their utilization.

$$P_B = \alpha_{B,f_B} \cdot C_{B,U} \cdot U_{B,U} + \beta_{B,f_B} \tag{3.1}$$

$$P_L = \alpha_{L,f_L} \cdot C_{L,U} \cdot U_{L,U} + \beta_{L,f_L} \tag{3.2}$$

$U_{B,U} = \frac{t_B}{t_F}$ and $U_{L,U} = \frac{t_L}{t_F}$ are the estimated utilization of the big and little cores that are actually used. The total power consumption can be estimated using equations 3.1 and 3.2. α_{B,f_B},β_{B,f_B}, α_{L,f_L}, and β_{L,f_L} are the linear-regression coefficients, which are determined by core type and frequency level.

3.1.3 Runtime Manager

The runtime manager consists of two parts. The main function periodically checks if the current performance satisfies the target performance. If unsatisfied, the main function invokes the search function to find a better system state. A better system state means that it satisfies or goes near to the target performance and maximizes the performance/power of the application. This is a similar metric to [6]. Note that performance is normalized performance, which is defined as $\min(g,h)/g$, where g is the target performance and h is the current performance. There is no benefit in overperformance. When the search function returns the better system state, the main function applies this to the next system state.

Algorithm 1 The main function of HARS

```
1: procedure HARSMAIN
2:   state ← initState
3:   setSysStateAndScheduleThreads(state)
4:   while true do
5:     hb ← getCurrentHeartBeat()
6:     if isAdaptPeriod(hb.index) then
7:       if |hb.rate − t.avg| > (t.max−t.min)/2 then
8:         state ← getNextSysState(hb, state, t, m, n, d)
9:         setSysStateAndScheduleThreads(state)
```

Algorithm 2 The search function of HARS

```
1: procedure GETNEXTSYSSTATE(hb, cs, t, m, n, d)
2:   initialize(ns)
3:   for i ∈ [cs.C_B − m, cs.C_B + n] do
4:     for j ∈ [cs.C_L − m, cs.C_L + n] do
5:       for k ∈ [cs.f_B − m, cs.f_B + n] do
6:         for l ∈ [cs.f_L − m, cs.f_L + n] do
7:           dist ← getDistance(i, j, k, l, cs)
8:           if dist > d then
9:             continue
10:          perf ← estimate_perf(i, j, k, l, hb, cs)
11:          power ← estimate_power(i, j, k, l)
12:          pp ← norm(perf)/power
13:          if perf ≥ t.min then
14:            if ns.perf ≥ t.min then
15:              if pp > ns.pp then
16:                updateCurrBestState(i, j, k, l, ns)
17:            else
18:              updateCurrBestState(i, j, k, l, ns)
19:          else
20:            if ns.perf < t.min then
21:              if perf > ns.perf then
22:                updateCurrBestState(i, j, k, l, ns)
23:   ns ← getBetterState(cs, ns)
24:   return ns
```

Algorithm 1 shows the main function, which receives heartbeats from the target self-adaptive application and checks if the current heartbeat is in the adaptation period (Line 6). The main function compares the current performance to the user-specified target performance and if the current performance does not meet the target, then it invokes the GetNextSysState function. This is the aforementioned search function.

Algorithm 2 shows the search function that is invoked by the main function to find the better system state. Its explorable area is determined by three configurable parameters m, n and d ($m \geq 0$, $n \geq 0$, $d > 0$). For example, if the current state of the big core count is C_B, then the search function can sweep the system state from $C_B - m$ to $C_B + n$. d is the Manhattan distance in four dimensional space (Line 7 in Algorithm 2). If the distance from the current state to the candidate state exceeds d, the search function skips that candidate system state. When these m, n and d parameters are larger, the search function can more exhaustively search the system state space with a higher runtime overhead.

The search function uses performance and power estimators to estimate the normalized performance/watt of each candidate (Lines 10, 11, and 12 in Algorithm 2). The search function finds the system state that satisfies the target performance at first. It then compares the estimated performance/power of

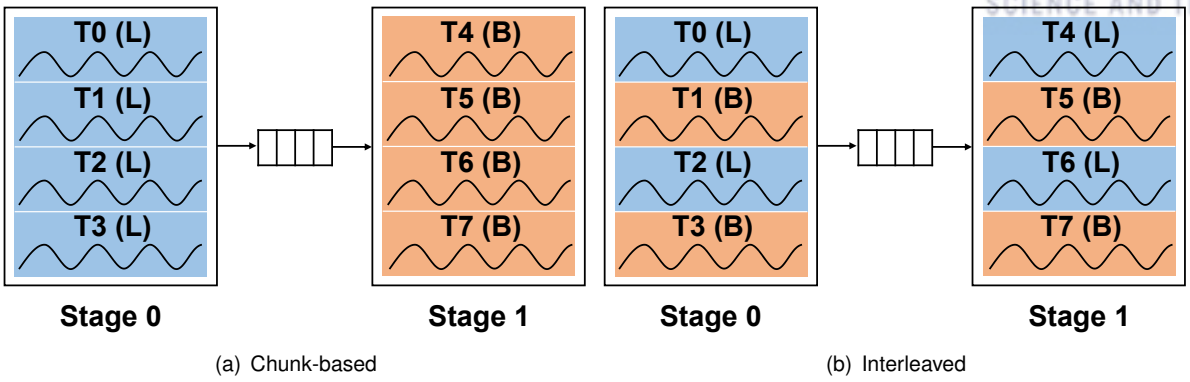


Figure 3.2: The HARS schedulers

each candidate and chooses the best system state with the highest performance/watt (Lines 10 – 22). So, although a certain state has the highest performance/watt, if it cannot satisfy the target, then another state that has a lower performance/watt than the certain state that achieves the target performance can be selected.

The runtime manager can explore the system states with two versions by configuring m, n, and d parameters differently. The `HARS-I` version is an incremental search version with $m = 1$, $n = 0$, and $d = 1$ (when the target application overperforms) or $m = 0$, $n = 1$, and $d = 1$ (when the target application underperforms). The `HARS-E` version is the exhaustive search version that can explore more system states than the `HARS-I` version. Each version has its pros and cons.

The `HARS-I` version has the advantage that it can explore system states with a low overhead because it searches a small set of system states. In addition, it may reduce the system oscillation by controlling the system state with $d = 1$. It has the disadvantage that it can be hard to approach the proper system state quickly and it may get stuck in a suboptimal system state.

The `HARS-E` version has the advantage that it can explore system states with a larger range, so it can approach the proper system state quickly and reduces the possibility of getting stuck in a suboptimal system state. However, it may incur a higher overhead than the `HARS-I` version and is potentially more vulnerable to system oscillation owing to the larger distance ($d = 7$).

The runtime manager has two types of schedulers. These schedulers use the `sched_setaffinity` function and get T_B and T_L parameters from the performance estimator (Table 3.1). We suppose that the all threads of the target self-adaptive application are ordered by the thread ID. The *chunk-based* scheduler assigns the consecutive T_L threads to the little cores $C_{L,U}$ first and the remaining threads T_B to the big cores $C_{B,U}$. This scheduler has the advantage that it can leverage the constructive cache sharing among the consecutive threads. However, it has the disadvantage that it can cause a performance imbalance on each thread if the target application uses parallelism such as pipelining with an asymmetric manner. Figure 3.2(a) shows an example of the chunk-based scheduler with a two-stage pipeline application. Four threads of the first stage are mapped on the little core and the second stage threads are mapped on big cores. In this case, the first stage can be a bottleneck and the whole application performance can be limited by this stage.

The *interleaving* scheduler can alleviate this problem. This scheduler maps threads to the big and little cores in an interleaved manner. Figure 3.2(b) shows the interleaving scheduler where each stage of pipeline application fairly assigns the cores of each core type. As each stage has the same resources, it

can eliminate the performance imbalance issue. However, it has the disadvantage that it may not exploit constructive cache sharing among threads.

3.1.4 Discussion

Although the current design of HARS shows good efficiency in our quantitative evaluation, HARS can be advanced by some techniques. First, the workload prediction model of HARS is a simple one that estimates that the amount of total unit of work is the same as the one observed before the heartbeat period. This scheme can be advanced with such technique as the Kalmanfilter [6], which dynamically predicts the uncertain workload in a more precise manner using educated guesses.

Second, the current version of HARS is designed with the assumption that total work is equally distributed on each thread in the self-adaptive application. If the application is parallelized (for example, a pipelining application with a different number of threads in each stage), this assumption does not apply. The interleaving scheduler may not fairly allocate the big core at each stage from this information. This problem can be solved using the thread hierarchy information. From this information, HARS can know the number of threads in each stage. So, the scheduler can assign cores to the threads more efficiently.

Third, the current design of HARS scheduler does not migrate the threads of the target application between the clusters until an adaptation period. We expect that efficient OS support for fine-grain inter-cluster scheduling [9] can lead to higher efficiency results. In our future work, we plan to investigate and enhance the current Linux HMP scheduler with the fine-grain and fair inter-cluster migration.

Fourth, if the current search algorithm of HARS gets stuck in a suboptimal point, it cannot escape the local optimal. It can be overcome by another algorithms (e.g., Tabu search [3]) that can escape the local optimal. These algorithms could be effective on applications that have a small variation of workloads, but it can lead to suboptimal results for applications with high workload variation.

Chapter 4

MP-HARS

4.1 MP-HARS Design & Implementation

We have shown HARS for a single application, but in the real-world, a system running with multiple applications is a common case. So, a runtime system that supports multiple applications in an efficient manner in a HMP system is needed. In this section, we will introduce the limitations to supporting multiple applications first and the solution to solve these limitations.

4.1.1 Limitation of the Naive Model that Supports Multiple Applications

If we plan to support multiple applications, there are two challenges. One is how to distribute the system resources to each application efficiently and the other is how to schedule the resources to its threads.

We will suggest the naive adaptation model for supporting multiple applications. In this model, all applications can share every system resource (core count, CPU frequency) with the Linux HMP scheduler. There are two limitations to this model. One is that each application has its own target performance but if all of applications share the system resources, some applications can use more resources than they need or suffer from a lack of resources. To prevent these suboptimal cases, the resource allocation of each application needs to be controlled. but it is not easy because of resource sharing. The other limitation is that the naive adaptation model cannot utilize the little core well by Linux HMP scheduler when the target application is CPU intensive. Especially, if the workload in the big cluster exceeds its capacity, it would be better to allocate the excess workload to the little cluster but the Linux HMP scheduler does not schedule like that.

This model is the incremental adaptation model that uses the list of system configurations sorted by performance score. The performance score can be calculated as $perfScore = C_B \cdot r_0 \cdot (f_B/f_0) + C_L \cdot (f_L/f_0)$, where r_0 is the performance ratio of the big and little core and f_0 is the baseline frequency. When the current performance is lower than the target, it chooses the state that has higher but nearest performance score to the current state and vice versa.

Each application has different adaptation period and behavior. When managing multiple applications, the decision of each application is not always same. There can be some applications that want to increase the system performance while others want to decrease it. In this case, the adaptation model should choose the one adaptation decision, increase, decrease or keep the system performance the same, because the model can control the core count and frequency but these components affect all applications in the system. If the core count is increased, the performance of all applications is increased. If the frequency is increased, the performance of all applications is increased and vice versa. There is only one choice although some applications do not want to.

When the adaptation decisions of applications conflict with each other, the conservative adaptation model is the simple guideline of the decision making that is used in the naive model. It gives top priority to achieving the target performance or keeping the satisfactory state. In addition, if the current performance of any application is not satisfied, there is no restriction on increasing system performance. If the adaptation decision is decreasing the system performance, some applications that do not satisfy the target could be harder to achieve the target performance. Alternatively, some applications that achieve the target performance can have lower performance than its target performance. Therefore, when the system performance is decreased, adaptation should be stopped for a certain period to collect new performance data from the new system state. Each application has its best system state but it is not easy to reach that

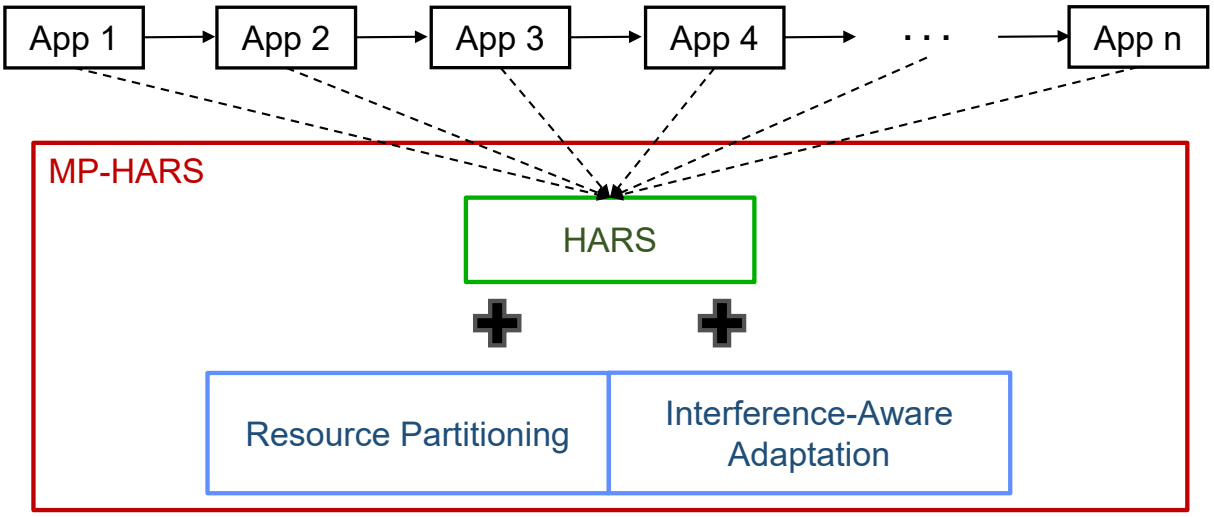


Figure 4.1: The overall architecture of MP-HARS

state because other applications also want to be at the best efficiency. Therefore, it is hard to set the system to run all applications with their best efficiency.

Linux HMP scheduler is the Global Task Scheduling model (GTS) that tracks the performance requirement for each individual thread. This scheduler has two thresholds, one is the *up migration threshold* where if the load average of thread mapped in the little cluster exceeds this threshold, the scheduler assigns this thread to the big cluster. The other is the *down migration threshold* whereby if the load average of the thread mapped in the big cluster is lower than this threshold, the scheduler assigns this thread to the little cluster. So if the CPU intensive applications run concurrently, the scheduler tends to migrate most threads to the big core cluster, which may lead to the load imbalance on each thread. From these limitations of the naive adaptation model, we propose MP-HARS, which gives more chance for the applications to approach their best system state.

4.1.2 MP-HARS Overview

The design goal of MP-HARS is similar to HARS: improve the performance/watt of each application while satisfying each target performance. MP-HARS considers the two aforementioned limitations. First is the limitation from resource sharing (cores and frequency) of all applications that restrict the adaptation behavior. Second is the load imbalance issue from the Linux HMP scheduler. In MP-HARS, the first limitation can be alleviated by preventing applications from using cores of other applications and giving more chance to control the frequency during the adaptation period. The second limitation can be solved by HARS for a single application. The *performance* estimator decides the best thread assignment, and sets the threads to run in little cores, if needed.

Figure 4.1 shows the overall architecture of MP-HARS. To support multiple applications, HARS needs two additional modules. One is the resource partitioning and the other is the interference-aware adaptation. Data from applications are managed with the linked-list data structure.

Algorithm 3 shows how the runtime manager accesses application data with the linked-list iteration. The *performance* estimation model and the *power* estimation model are the same as HARS for a single

Algorithm 3 The iterate function of MP-HARS

```
1: procedure ITERATENODES
2:   p ← listHead
3:   while p != NULL do
4:     appData ← readCurrentNodeData()
5:     p ← getNextNode()
6:     hb ← getCurrentHeartBeat(appData)
7:     if newHB(hb) then
8:       if appData→freezing_cnt_b > 0 then
9:         appData→freezing_cnt_b- -
10:      if appData→freezing_cnt_l > 0 then
11:        appData→freezing_cnt_l- -
12:      if checkFreezingCnt(bigCluster) then
13:        setFrozenState(bigCluster)
14:      if checkFreezingCnt(littleCluster) then
15:        setFrozenState(littleCluster)
16:      if isAdaptPeriod(hb.index) then
17:        if |hb.rate − (appData→t.avg)| > ((appData→t.max)−(appData→t.min))/2 then
18:          freeCoreCnt ← checkFreeCore()
19:          controllableCluster ← checkClusterControllable()
20:          state ← getNextSysState(hb, state, t, m, n, d, freeCoreCnt, controllableCluster)
21:          allocatedCoreSet ← getAllocatableCoreSet()
22:          setSysStateAndScheduleThreads(state, allocatedCoreSet)
23:    if FreqIsDecreased(bigCluster) then
24:      setFreezingCnt(bigCluster)
25:    if FreqIsDecreased(littleCluster) then
26:      setFreezingCnt(littleCluster)
```

application. From Line 16, the adaptation algorithm looks similar to the main function of HARS (Algorithm 1) for a single application but it uses two additional data checking processes one is finding the free cores that are not allocated to any application (Line 18) and the other is configurable information about cluster frequency (Line 19). The freeCoreCnt is the number of cores that any application does not use and the controllableCluster is the state that describes if the frequency of the clusters can be controlled. After these processes, the runtime manager invokes the search function, which explores the system state to find the best next system state (Line 20). However, the searchable core count is limited by freeCoreCnt and the frequency controllable type of cluster is limited by controllableCluster variables. When the next system state is chosen, the runtime manager scans the allocatable cores on system and makes a cpu set (Line 21). Each iteration, the runtime manager checks if applications are in an adaptation period one by one.

4.1.3 Resource partitioning

The resource partitioning allows applications to use their cores and controls the frequency individually. However, it is not always available to control the frequency level individually. Changing the frequency level is only possible at the cluster level so cores in the same cluster share the same frequency level. It is possible to control the frequency level freely when the applications do not use the cores on the same cluster. Therefore, there is more chance of controlling the frequency than the naive version, as mentioned before. On the other hand, if the applications use the core of same cluster, the frequency cannot easily be

Variable	*Description*
int nprocs_b	number of assigned big cores
int nprocs_l	number of assigned little cores
int use_b_core[4]	array for assigned big cores
int use_l_core[4]	array for assigned little cores
int64_t adaptation_index	adaptation heartbeat index
double heartbeat_rate	performance of application
int freezing_cnt_b	number of heartbeats to wait big frequency is controllable
int freezing_cnt_l	number of heartbeats to wait little frequency is controllable

Table 4.1: Per application data structure

Variable	*Description*
int frozen_flag	frozen state of cluster
int free_core[4]	array for free/unfree core
int nfreq	frequency level of cluster

Table 4.2: Per cluster data structure

controlled. If the applications share the same cluster and try to control the frequency level, we should use the conservative adaptation model.

Resource partitioning enables the application to adapt individually. In the naive adaptation model, if some applications have lower performance than the target, the system performance cannot be decreased in any case although another application wants to decrease the performance. On the other hand, MP-HARS can decrease the application performance if it has a margin to decrease the core count or frequency level of the cluster that the other application does not use.

To control the application resource individually, the runtime manager needs per application data structure. We can see the structure in Table 4.1. Data from each application is managed by the linked list. In addition, one node has one data structure of the application. Algorithm 4 shows how to allocate cores to each application individually. First, it checks the decreased core count from the previous adaptation and decides how many cores to free. We use two data arrays for managing the individual core allocation. One is the free core index information of the cluster (appData→use_coretype_core[4]) and the other is the information about the core index assigned to the application (cluster.free_core[4]). When the runtime manager newly assigns a core to an application, it looks up the assigned core information to which one is already used. If a used core exists, it does not need to newly assign another core because it wants to minimize the thread migration to another core. If the new core count is lower than the used core count, the left number of cores will be offered by the free cores. For example, let us assume that there are two applications running concurrently. $Application_A$ was assigned to $bigcore_{0-1}$ and $Application_B$ was assigned to $littlecore_{0-1}$. If $Application_B$ needs more resources and wants to use the big core, it cannot get $bigcore_{0-1}$. Instead, it can get $bigcore_{2-3}$. Because, $bigcore_{0-1}$ are used by $Application_A$ and $bigcore_{2-3}$ are not used by any applications. $bigcore_{2-3}$ are called the free core.

4.1.4 Interference-aware adaptation

The interference-aware adaptation considers the effect of other applications' performance changing when controlling a shared system component and chooses the safest way to prevent other applications from

violating the target performance. This adaptation model is applied when the applications control the frequency of the same cluster.

If the frequency of a certain cluster is changed by one application, the performance of the other applications that mapped in that cluster can also be changed. Especially, if the frequency is decreased by one application and the other application tries to adapt but it does not have enough time to collect performance data on new system state, the adaptation behavior could be inaccurate and unsafe to violate or far away from the target performance. Therefore, if the frequency is decreased, it needs to force all affected applications to wait until all of them spend enough time to get reliable performance data.

The *freezing count* is the per application data for the number of heartbeats to wait enough time to collect the performance data of new system state. An application has two freezing count, is for the big cluster and the other for the little cluster. The *frozen state* is the per cluster state that cannot decrease the frequency of the frozen cluster. The frozen state is set when the runtime manager scans the freezing count of each application and if any application has a freezing count that bigger than 0 (Lines 12 – 15 in Algorithm 3).

The frozen state can be unfreezed in two cases. The first is if the system performance needs to be increased and the second is if the freezing count of all applications that is affected by the decreased frequency is 0. Table 4.3 shows how the frozen state is decided to freeze or unfreeze. *AppInPeriod* means the performance satisfaction state of an application that is in the adaptation period. To save the information about the available core index and the freezing state of each cluster, we need per cluster data structure, which can be seen in Table 4.2.

Lines 8 – 15 and 23 – 26 in Algorithm 3 show how to manage the frozen state. First, when the new heartbeat of the application comes in, it checks the freezing count of each cluster type and decreases the freezing count if it is higher than 0 (Lines 8 – 11). Second, the runtime manager checks the freezing count of each application and if any application has a freezing count bigger than 0, it sets the frozen state on the cluster that has a nonzero freezing count (Lines 12 – 15). After the adaptation, runtime manager checks if frequency is decreased (Lines 23 and 25). If it is decreased, it sets the freezing count of each application that uses the cluster whose frequency was decreased (Lines 24 and 26).

To summarize, the MP-HARS design looks like each application is managed by its own HARS. It allows applications to have their own cores that the other application cannot use. However, the frequency level cannot be controlled individually. If the frequency level changing interferes with other applications, it uses a conservative approach with the frozen state concept.

$AppInPeriod$	$TheOthers$	$FrozenState$	$StateDecision$	$FreezeDecision$
Underperf	Underperf	FREEZE	INC	**UNFREEZE**
		UNFREEZE	INC	KEEP
	Achieve	FREEZE	INC	**UNFREEZE**
		UNFREEZE	INC	KEEP
	Overperf	FREEZE	INC	**UNFREEZE**
		UNFREEZE	INC	KEEP
Achieve	Underperf	FREEZE	KEEP	KEEP
		UNFREEZE	KEEP	KEEP
	Achieve	FREEZE	KEEP	KEEP
		UNFREEZE	KEEP	KEEP
	Overperf	FREEZE	KEEP	KEEP
		UNFREEZE	KEEP	KEEP
Overperf	Underperf	FREEZE	INC	KEEP
		UNFREEZE	KEEP	KEEP
	Achieve	FREEZE	INC	KEEP
		UNFREEZE	KEEP	KEEP
	Overperf	FREEZE	INC	KEEP
		UNFREEZE	DEC	**FREEZE**

Table 4.3: State & Freeze decision Table

Algorithm 4 The core allocation function of MP-HARS

```
1: procedure GETALLOCATABLECORESET
2:   allocatedBigCoreCnt ← 0
3:   allocatedLittleCoreCnt ← 0
4:   if appData→decBigCoreCnt > 0 then
5:     for i ∈ [0,MaximumBigCoreCount] do
6:       if appData→use_b_core[i] == USE then
7:         bigCluster.free_core[i] ← FREE
8:         appData→use_b_core[i] ← UNUSE
9:         appData→decBigCoreCnt- -
10:        if appData→decBigCoreCnt == 0 then
11:          break
12:  if appData→decLittleCoreCnt > 0 then
13:    for i ∈ [0,MaximumLittleCoreCount] do
14:      if appData→use_l_core[i] == USE then
15:        LittleCluster.free_core[i] ← FREE
16:        appData→use_l_core[i] ← UNUSE
17:        appData→decLittleCoreCnt- -
18:        if appData→decLittleCoreCnt == 0 then
19:          break
20:  for i ∈ [0,MaximumBigCoreCount] do
21:    if allocatedBigCoreCnt < appData→nprocs_b then
22:      if appData→use_b_core[i] == USE then
23:        bigCluster.free_core[i] ← NOT_FREE
24:        cpuMask ← CPUSet(i+bigStartIndex, cpuMask)
25:        allocatedBigCoreCnt++
26:  for i ∈ [0,MaximumBigCoreCount] do
27:    if allocatedBigCoreCnt < appData→nprocs_b then
28:      if bigCluster.free_core[i] == FREE then
29:        bigCluster.free_core[i] ← NOT_FREE
30:        appData→use_b_core[i] ← USE
31:        cpuMask ← CPUSet(i+bigStartIndex, cpuMask)
32:        allocatedBigCoreCnt++
33:  for i ∈ [0,MaximumLittleCoreCount] do
34:    if allocatedLittleCoreCnt < appData→nprocs_l then
35:      if appData→use_l_core[i] == USE then
36:        littleCluster.free_core[i] ← NOT_FREE
37:        cpuMask ← CPUSet(i, cpuMask)
38:        allocatedLittleCoreCnt++
39:  for i ∈ [0,MaximumLittleCoreCount] do
40:    if allocatedLittleCoreCnt < appData→nprocs_l then
41:      if littleCluster.free_core[i] == FREE then
42:        littleCluster.free_core[i] ← NOT_FREE
43:        appData→use_l_core[i] ← USE
44:        cpuMask ← CPUSet(i, cpuMask)
45:        allocatedLittleCoreCnt++
46:  return cpuMask
```

Chapter 5

Evaluation

5.1 Evaluation: HARS

5.1.1 Methodology

Our evaluation was conducted on a real full heterogeneous multi-processing (HMP) system. The ODROID-XU3 embedded board uses the Samsung Exynos 5422 processor with ARM's big.LITTLE architecture [7]. This is an octa-core processor that consists of four big cores and four little cores. The big core is a complex and out-of-order processor called Cortex-A15 and the little core is a simple and in-order processor called Cortex-A7. We assume that performance ratio between big and little core can be calculated as $r_0 = \frac{S_{B,f_0}}{S_{L,f_0}} = \frac{3}{2}$. This assumption comes from the instruction width ratio of big (3) and little core (2). This performance ratio applies on performance estimator of HARS.

We use Xubuntu 14.04 and Linux kernel version 3.10.51, which implements the HMP scheduler (GTS). The frequency range of big and little core are 0.8 – 1.6 GHz and 0.8 – 1.3 GHz, respectively. The frequency level can be scaled by DVFS. The board is equipped with a power sensor that can measure the voltage and current of the big and little core clusters with a 263,808 μs sampling period. Linear regression models of the power estimator were constructed from this data that was collected from the sensors.

To measure the effectiveness of HARS, we use six benchmarks of the PARSEC benchmark Suite (`blackscholes` (BL), `bodytrack` (BO), `facesim` (FA), `ferret` (FE), `fluidanimate` (FL), and `swaptions` (SW)) as target self-adaptive multithreaded applications [1]. We use the native inputs of each benchmark except for `swaption` (i.e., `-ns 12800 -sm 10000`) to increase the number of heartbeat intervals.

We evaluate each PARSEC benchmark with the following five versions. The *baseline* version runs at the maximum core count and frequency level scheduled by the Linux HMP scheduler.

The *static optimal* version runs with the optimal number of cores and frequency level determined by the offline simulations. The simulations are based on the algorithms discussed in Section 3.1 that sweeps all available system states and estimates the performance/watt. It also scheduled by the Linux HMP scheduler.

The `HARS-I` and `HARS-E` versions use the Application Heartbeat API [4] to set the user-specified target performance of each benchmark and get the current performance. These versions dynamically control the system state with the search function of HARS and are scheduled by the chunk-based scheduler. The interleaving scheduler is implemented on the `HARS-EI` version, which is similar to the `HARS-E` version.

We set the *default performance target* to half (50%±5%) of the maximum achievable performance of each benchmark. This target is similar to the previous work [6]. Futhermore, to investigate the sensitivity of target performance, we also evaluate the *high performance target* of three quarters (75%±5%) of maximum achievable performance of each benchmark. We set the thread-count parameter `n` of each PARSEC benchmark runs with the total number of cores (8).

5.1.2 Experimental Results

We will evaluate HARS with the following purposes. First, we demonstrate the effectiveness of HARS by evaluating the performance/watt improvement of each benchmark comparing to the baseline and the static optimal versions. Second, we monitor the changing of effectiveness between the chunk-based and interleaving schedulers of HARS. Third, we investigate the sensitivity of target performance in HARS.

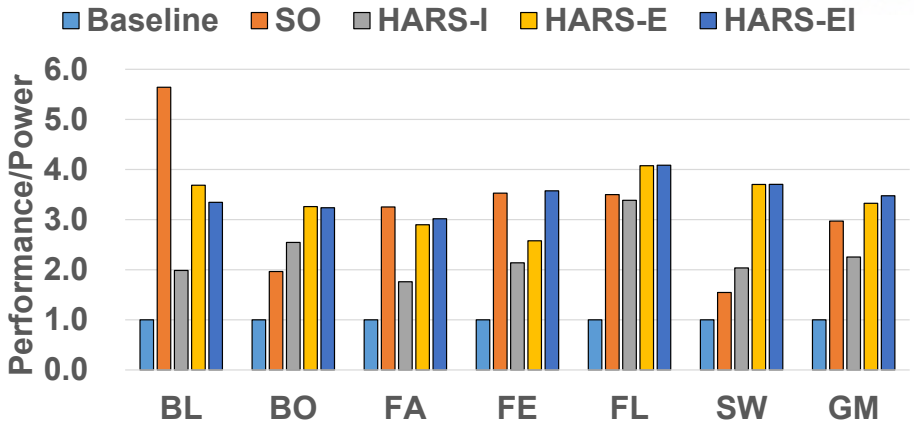


Figure 5.1: Performance/watt results in default-perf. target

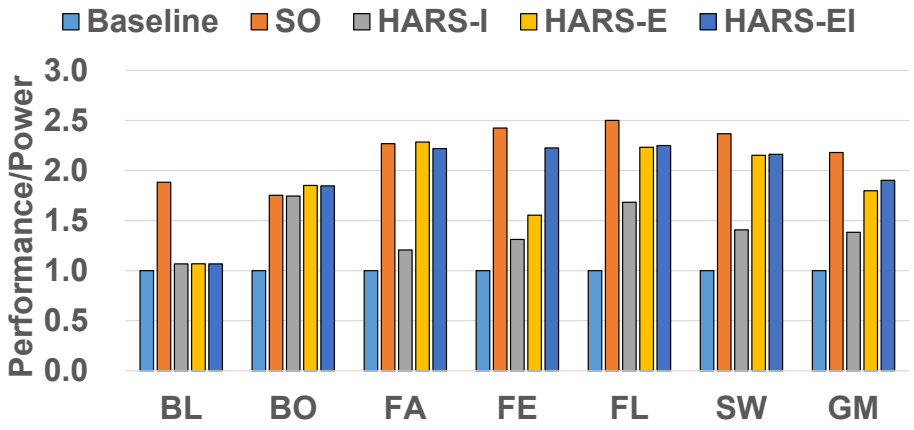


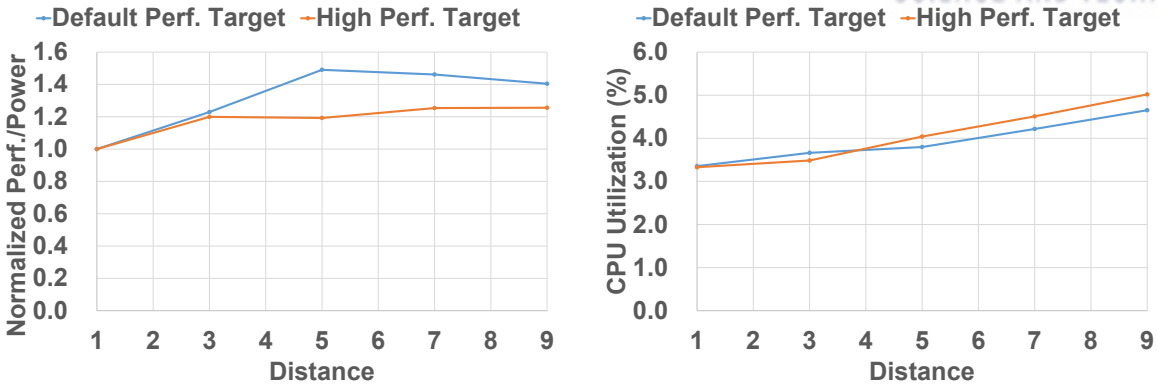
Figure 5.2: Performance/watt results in high-perf. target

Fourth, we checked the efficiency and the runtime overhead of HARS when sweeping the range of exploring system state space.

Figure 5.1 shows the performance/watt results of the default performance target. Each bar shows the aforementioned five versions of each benchmark normalized to the baseline version and the rightmost bar shows the geometric mean. The baseline version shows the lowest results compared to another version because this version largely overperforms the target and consumes excessive power. The static optimal (SO) version significantly outperforms the baseline version by using the optimal number of cores and frequencies.

The `HARS-I` version largely outperforms the baseline version through the dynamic adaptation. However, the static optimal version shows the better performance/watt in many cases because of slow adaptation and the possibility of getting stuck in the local optimal.

The `HARS-E` version significantly outperforms the `HARS-I` version owing to faster adaptation than the `HARS-I` version and lower possibility of getting stuck in the local optimal by exploring the system-state space more exhaustively. The result is comparable to the static optimal version, but the results of `blackscholes` and `ferret` benchmarks are largely outperformed by the static optimal



(a) Performance/watt

(b) CPU utilization

Figure 5.3: The efficiency and overhead versus the explored space size

version. In `blackscholes` benchmark, inaccurate big and little performance ratio leads to suboptimal result. HARS assumes 1.5 as a performance ratio between the big and little cores (r_0). We evaluate the performance ratio of the `blackscholes` benchmark and it shows same the performance on the big and little cores ($r_0 = 1.0$). In our a future work, we plan for HARS to update the performance ratio in real time.

The `ferret` benchmark is a 6-stage pipeline parallelism application that is vulnerable to performance imbalance (Section 3.1.3) of threads because the `HARS-E` version uses the chunk-based scheduler, it maps some stages only on little cores, which may cause a bottleneck, but the `HARS-EI` version can mitigate this problem.

The results show that the `HARS-EI` version of `ferret` benchmark shows comparable efficiency to the static optimal version. `HARS-EI` uses an interleaving scheduler that mitigates the performance imbalance issue by fairly scheduling the big and little cores to the threads of each stage.

Figure 5.1 shows that the `HARS-EI` version has high efficiency on most of the benchmarks, slightly outperforming both the static optimal and `HARS-E` versions on geometric mean.

Figure 5.2 also shows that the HARS versions have significantly better performance/watt than the baseline version and are comparable to the static optimal version. However, we observe some different trends comparing to the *default performance target*. First, the efficiency gain of the static optimal and HARS versions over the baseline version are smaller than the efficiency gains with the `default performance target`. This is because the energy benefit chance is smaller from the maximum core counts and frequency than the *default performance target*.

Second, the `blackscholes` benchmark of HARS result shows much lower performance/watt than the static optimal version because HARS has few chances for adaptation and its wrong performance ratio (r_0) leads suboptimal adaptation. This benchmark workload variation is stable that has no additional chance to step out of the local optimal state. Actually, the settled system state of the `blackscholes` benchmark is a suboptimal state. In our future work, we plan to integrate more advanced search algorithms to explore the system-state space and have the escaping local optimal solution with wide coverage and high efficiency. Nevertheless, the exhaustive search version of HARS significantly outperform the baseline version and was comparable to the static optimal version.

We study the efficiency and the runtime overhead of HARS when sweeping the range of exploring system state space. We sweep the distance parameter (d) from 1 to 9 with a step of 2 to control the size of exploring system state space. Figures 5.3(a) and 5.3(b) show the performance/watt and the CPU utilization of HARS that was run with the `HARS-EI` version for all the benchmarks. Figure 5.3(a) shows the geometric mean of the benchmarks results and the performance/watt is normalized to $d = 1$. Figure 5.3(b) shows the average CPU utilization of the benchmarks.

We can see that when the d value increases, the efficiency of HARS also increases. However, we also observe a certain threshold that exist on $d = 5$ in Figure 5.3(a). The CPU utilization of HARS (i.e., overhead) also increases in larger d value. Because the runtime manager explores more system states. As a result, we can see that the runtime overhead of HARS is small (i.e., the average CPU utilization is under 6% with $d = 9$). Therefore, HARS effectively improves the efficiency of the target self-adaptive multithreaded applications with a small runtime overhead.

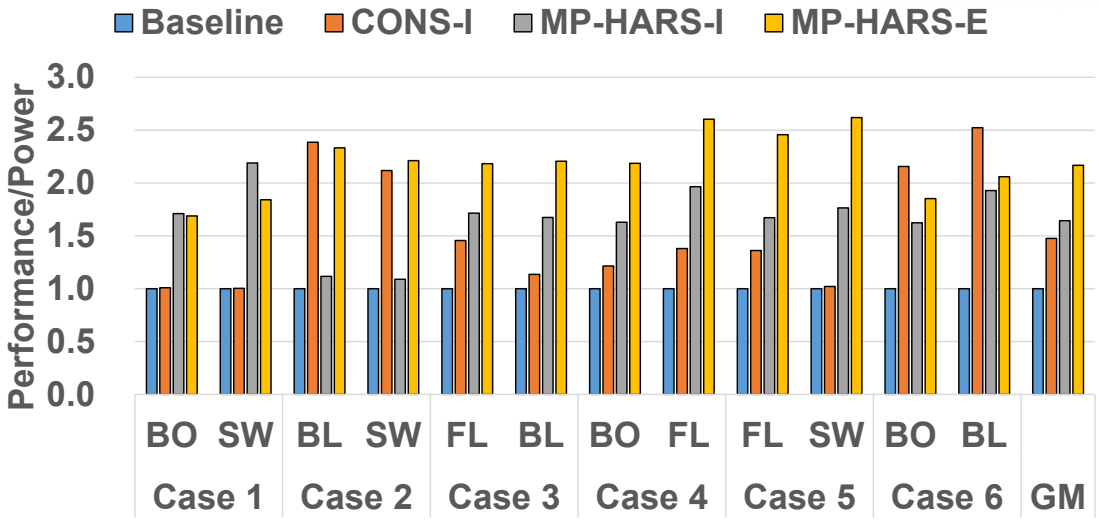


Figure 5.4: Performance/watt results about multi-application

5.2 Evaluation: MP-HARS

5.2.1 Methodology

To demonstrate the effectiveness of HARS with multi-applications, we use four PARSEC benchmarks (`blackscholes` (BL), `bodytrack` (BO), `fluidanimate` (FL), `swaptions` (SW)) and evaluate the six cases (BO-SW, BL-SW, FL-BL, BO-FL, FL-SW, BO-BL). The applications start to run at the same time. We specify the target performance level to half ($50\%\pm5\%$) of the maximum achievable performance. We evaluate each case on the following four versions. The baseline version is the Linux HMP scheduler, which runs with the maximum core counts and frequency level. The `CONS-I` version is a conservative incremental adaptation version that uses the aforementioned naive adaptation model. It is an incremental adaptation model and does not estimate the performance or power consumption. Instead, it chooses the candidate system state that makes the smallest system performance change. In addition, it allows applications to share all of cores and frequency levels. It uses the interference-aware adaptation policy.

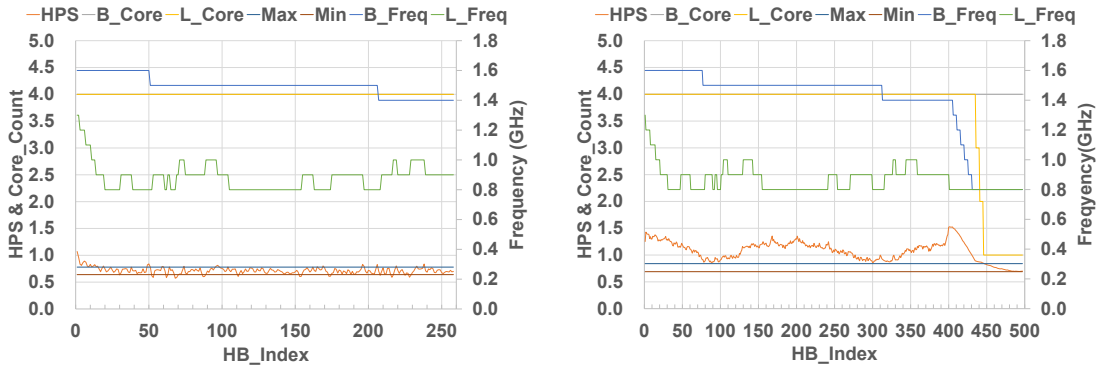
The `MP-HARS-I` and `MP-HARS-E` do not allow applications to share their core, but they can share a cluster frequency level. Frozen state policy is applied when controlling the frequency level that causes a interference on another application. These two versions use HARS estimation models to find the optimal allocation state. `MP-HARS-I` is an incremental search version that explores neighbor states with distance 1. `MP-HARS-E` is an exhaustive search version that explores neighbor states with distance 7 ($m=4, n=4, d=7$).

5.2.2 Experimental Results

In evaluation, we will focus on improvement from the resource partitioning, interference-aware adaptation and thread assignment. Figure 5.4 shows the performance/watt results with various combination of benchmarks. The rightmost bar is the geometric mean of the all cases. The `MP-HARS-E` version achieves better performance per power than the baseline and `CONS-I` versions by 217% and 46%. In case 4, we

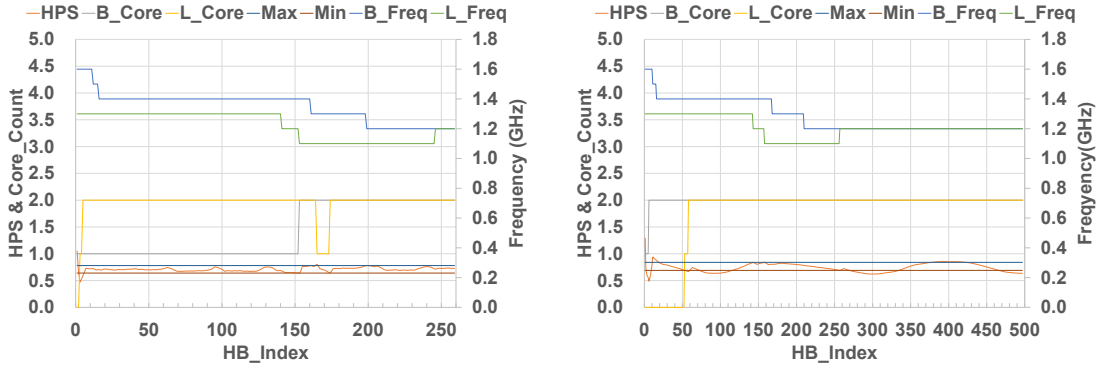
can see that the MP-HARS-I and the MP-HARS-E versions show better result compared to that of the CONS-I version. Figure 5.5(a) shows that the bodytrack benchmark achieves target performance well. However, in Figure 5.5(b), the performance of the fluidanimate benchmark largely exceeds the target performance. In range of HB_index 160 to 240, it does not try any adaptation because the bodytrack benchmark is achieving the target performance. The CONS-I version does not decrease the system performance if the certain application achieves the target. It leads to a suboptimal result. On the other hand, in Figure 5.6, we can see that the two applications achieve their target well because the MP-HARS-I version can manage the resources of each application so the adaptation chance of the fluidanimate benchmark is not limited by another application. Therefore, the MP-HARS-I version shows better result than that of the CONS-I version. The MP-HARS-E version can explore a larger system state space than the MP-HARS-I version. The MP-HARS-I version can only change the system components one by one. When the target is achieved in a certain system state, it does not try to adapt anymore. In this case, there could be a more optimal point than the current point. The MP-HARS-E version can compare the performance per power of more system states so it has lower possibility of getting stuck in a local optimal point compared to the MP-HARS-I version. In Figure 5.7, the bodytrack benchmark of the MP-HARS-E version shows that it uses the little core preferentially and does not use any big core. The fluidanimate benchmark of the MP-HARS-E version uses four big cores but the big cluster frequency is relatively low to maximum. When we approximately average out the used system resources, the MP-HARS-I version uses three big cores with 1.3GHz and four little cores with 1.2 GHz. On the other hand, the MP-HARS-E version uses three big cores with 0.9 GHz and four little cores with 1.3 GHz, which is expected to consume lower power.

In case 6, the CONS-I version shows better result than that of the HARS adaptation versions. This is because, the blackscholes benchmark has a phase for collecting input data. During this phase, a heartbeat is not emitted. Therefore, in the early phase, only the bodytrack benchmark executes the work and emits the heartbeats. In the CONS-I version, it decreases the frequency level of two clusters first, and then achieves the target performance. After the early phase, the blackscholes benchmark starts to emit heartbeats but it already nearly approaches the system state that can have good performance per power. In the MP-HARS-E version, as we mentioned before, the blackscholes benchmark starts later. The bodytrack benchmark starts in advance and the runtime manager sets this benchmark to run at good efficiency. Then, the blackscholes benchmark starts later but it can only use the cores that are not allocated to the bodytrack benchmark. From this adaptation behavior, there is no free little core to assign on the blackscholes benchmark. Therefore, the blackscholes benchmark has no choice but to use the big cores. As we mentioned in Section 5.1, the blackscholes benchmark shows the same performance in the big and little cores, but the big core consumes more power than the little core. If the blackscholes benchmark mainly uses the big core, it could be a suboptimal point. From this result, we can expect that this suboptimal result can be alleviated by the information about the big and little core performance ratio.



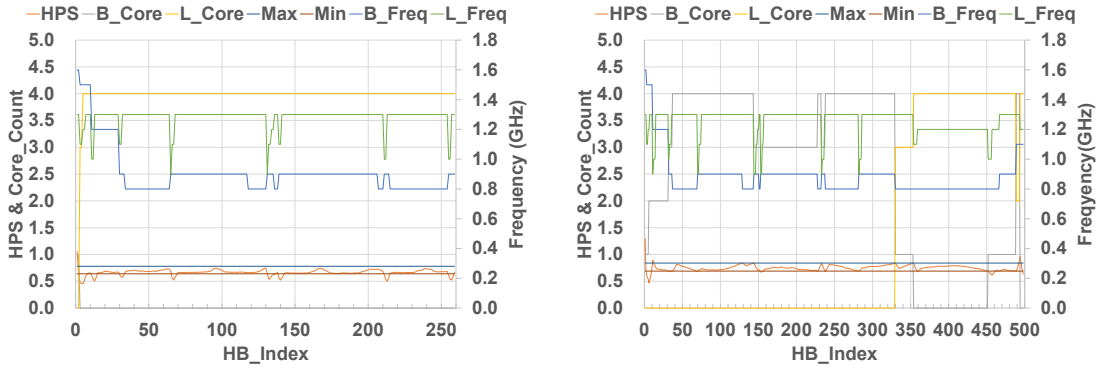
(a) BO (b) FL

Figure 5.5: Behavior graph of case 4 in CONS-I



(a) BO (b) FL

Figure 5.6: Behavior graph of case 4 in MP-HARS-I



(a) BO (b) FL

Figure 5.7: Behavior graph of case 4 in MP-HARS-E

Chapter 6

Related Work

6.1 Related Work

Prior work has extensively investigated the runtime and OS support for self-adaptive computing [4, 5, 6, 10, 13]. In [4], Hoffmann et al. proposed the *Application Heartbeats* framework, which provides simple yet effective computing [4]. Runtime and OS support for self-adaptive computing have been investigated in [5, 6, 13]. Their difference from the ours is that they investigated the self-adaptive system support for symmetric multi-processing (SMP) systems. Similar research to our work can be seen in [10], which investigated the OS support for self-adaptive single-threaded applications on HMP systems.

Prior work has also extensively investigated the architectures [8], scheduling techniques [9, 12, 15], and performance and power models [11, 14] for HMP. However, we targeted multithreaded applications on the HMP system to enhance the efficiency by self-adaptive computing. However, we believe that HARS can be advanced with more accurate performance and power models [11, 14] and scheduling techniques [9, 12, 15].

Chapter 7

Conclusions

7.1 Conclusions

In this work, we propose HARS, a heterogeneity-aware runtime system for self-adaptive multithreaded applications. HARS enables the target self-adaptive application to define its target performance and periodically checks if the application-level performance satisfies the target performance. If unsatisfied, HARS dynamically changes the system state to improve the efficiency of the target application on HMP systems, while satisfying the performance target. As an extended version of HARS, we also propose MP-HARS, which supports multiple applications. MP-HARS monitors each application's performance and compares to target performance one by one. Each application uses its own cores and does not share with other applications. Through our evaluation, HARS significantly outperforms the baseline version and are comparable to the static optimal version. MP-HARS largely outperforms the baseline version. In our future work, we plan to improve MP-HARS in several ways, such as implementing more advanced search algorithms to explore the large system state space more efficiently and more quickly find the best system state that can escape from the suboptimal point and researching more efficient and flexible policies to share system resources across multiple applications.

REFERENCES

[1] C. Bienia, S. Kumar, J. P. Singh, and K. Li. “The PARSEC Benchmark Suite: Characterization and Architectural Implications”. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 2008.

[2] *big.LITTLE Technology*. `http://www.arm.com/products/processors/technologies/biglittleprocessing.php`.

[3] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[4] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. “Application Heartbeats: A Generic Interface for Specifying Program Performance and Goals in Autonomous Computing Environments”. In: *Proceedings of the 7th International Conference on Autonomic Computing*. ACM, 2010.

[5] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. “Dynamic Knobs for Responsive Power-aware Computing”. In: *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2011.

[6] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. “A Generalized Software Framework for Accurate and Efficient Management of Performance Goals”. In: *Proceedings of the Eleventh ACM International Conference on Embedded Software*. IEEE Press, 2013.

[7] B. Jeff. “Big.LITTLE system architecture from ARM: saving power through heterogeneous multi-processing and task context migration”. In: *Proceedings of the 49th Annual Design Automation Conference*. 2012.

[8] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen. “Single-ISA Heterogeneous Multi-Core Architectures: The Potential for Processor Power Reduction”. In: *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

[9] T. Li, P. Brett, R. Knauerhase, D. Koufaty, D. Reddy, and S. Hahn. “Operating system support for overlapping-ISA heterogeneous multi-core architectures”. In: *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. 2010.

[10] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin. “Hierarchical Power Management for Asymmetric Multi-core in Dark Silicon Era”. In: *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013.

[11] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin. “Power-performance Modeling on Asymmetric Multi-cores”. In: *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. IEEE Press, 2013.

[12] D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. “HASS: A Scheduler for Heterogeneous Multicore Systems”. In: *SIGOPS Oper. Syst. Rev.* 43.2 (2009).

[13] F. Sironi, D. B. Bartolini, S. Campanoni, F. Cancare, H. Hoffmann, D. Sciuto, and M. D. Santambrogio. “Metronome: Operating System Level Performance Management via Self-adaptive Computing”. In: *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012.

[14] K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. “Scheduling Heterogeneous Multi-cores Through Performance Impact Estimation (PIE)”. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. IEEE Computer Society, 2012.

[15] K. Van Craeynest, S. Akram, W. Heirman, A. Jaleel, and L. Eeckhout. “Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores”. In: *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*. IEEE Press, 2013.